CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros Informáticos

TRABAJO FIN DE GRADO

Desarrollo de funcionalidad para un marco orientado al razonamiento sobre algoritmos criptográficos

Autor: Guillermo Ramos Gutiérrez
Director: Pablo Nogueira Iglesias

MADRID, JUNIO DE 2014

Resumen

La sociedad depende hoy más que nunca de la tecnología, pero la inversión en seguridad es escasa y los riesgos de usar sistemas informáticos son cada día mayores. La criptografía es una de las piedras angulares de la seguridad en este ámbito, por lo que recientemente se ha dedicado una cantidad considerable de recursos al desarrollo de herramientas que ayuden en la evaluación y mejora de los algoritmos criptográficos. EasyCrypt es uno de estos sistemas, desarrollado recientemente en el Instituto IMDEA Software en respuesta a la creciente necesidad de disponer de herramientas fiables de verificación de criptografía. A lo largo de este trabajo se abordará el diseño e implementación de funcionalidad adicional para EasyCrypt.

En la primera parte de documento se discutirá la importancia de disponer de una forma de especificar el coste de algoritmos a la hora de desarrollar pruebas que dependan del mismo, y se modificará el lenguaje de EasyCrypt para permitir al usuario abordar un mayor espectro de problemas.

En la segunda parte se tratará el problema de la usabilidad de EasyCrypt y se intentará mejorar dentro de lo posible desarrollando una interfaz web que permita usar el sistema fácilmente y sin necesidad de tener instaladas todas las herramientas que necesita EasyCrypt.

Abstract

Today, society depends more than ever on technology, but the investment in security is still scarce and the risk of using computer systems is constantly increasing. Cryptography is one of the cornerstones of security, so there has been a considerable amount of effort devoted recently to the development of tools oriented to the evaluation and improvement of cryptographic algorithms. One of these tools is EasyCrypt, developed recently at IMDEA Software Institute in response to the increasing need of reliable cryptography verification tools. Throughout this document we will design and implement two different EasyCrypt features.

In the first part of the document we will consider the importance of having a way to specify the cost of algorithms in order to develop proofs that depend on it, and then we will modify the EasyCrypt's language so that the user can tackle a wider range of problems.

In the second part we will assess EasyCrypt's poor usability and try to improve it by developing a web interface which enables the user to use it easily and without having to install the whole EasyCrypt toolchain.

Índice general

Capítulo 1

INTRODUCCIÓN

Con el paso del tiempo, la sociedad está aumentando progresivamente su dependencia de los sistemas informáticos, tanto a nivel local como global (internet). La gente gestiona sus cuentas bancarias por internet, está permanentemente en contacto con sus conocidos a través de aplicaciones de mensajería instantánea y almacena en *la nube* gran cantidad de material personal. Esta dependencia trae consigo la necesidad de tener unas ciertas garantías de seguridad (confidencialidad, disponibilidad, integridad) tanto a la hora de transmitir la información como de almacenarla. Construir un sistema de seguridad no es una tarea fácil, ya que a menudo consta de muchos componentes que han de ser examinados individualmente de forma rigurosa para poder afirmar que el conjunto es seguro.

Uno de los componentes básicos de prácticamente cualquier sistema de seguridad es la criptografía. Su utilidad no se limita a impedir que una persona no autorizada con acceso a datos privados pueda entenderlos (cifrado), sino que también cubre otras necesidades como la de verificación de identidad (autenticación) o la de poder garantizar de que una información no ha sido alterada (integridad). Poniendo como ejemplo el caso de una persona accediendo via web a la página web de su banco para realizar una transferencia, la criptografía juega un papel central a la hora de garantizar la seguridad de la operación: un protocolo de transporte seguro empleará un esquema de criptografía asimétrica para garantizar al cliente que el servidor es quien dice ser, un algoritmo de cifrado para hacer imposible que cualquier intermediario pueda interceptar la comunicación, y un código de autenticación de mensaje para verificar que los datos no se modifican mientras están viajando.

Por lo general, estas primitivas criptográficas basan su seguridad en la dificultad computacional que tiene resolver ciertos problemas. Por ejemplo, uno de los protocolos más utilizados para autenticar servidores, RSA, se basa en que un atacante que intente suplantar al servidor en cuestión tendrá antes que descomponer un número en dos factores primos, problema que se considera inabordable para números lo suficientemente grandes (con la capacidad computacional actual). Como resulta-

do, normalmente no se habla en términos de seguridad absoluta, sino de seguridad **condicionada a la capacidad computacional del atacante**.

Para estar seguros de que los mecanismos criptográficos realmente cumplen con su cometido, es necesario someterlos a un proceso de verificación riguroso. El objetivo es poder razonar sobre ellos y demostrar que cumplen ciertas propiedades de seguridad que consideramos favorables. En la mayoría de los casos se intenta encontrar una **reducción** [1] del criptosistema a un problema difícil computacionalmente, de forma que un ataque exitoso contra el primero implicaría poder resolver de forma eficiente el segundo (ver figura 1.1). Continuando el ejemplo anterior, RSA se considera seguro porque la existencia de un ataque eficiente contra el criptosistema implicaría también la existencia de una forma eficiente de factorizar enteros (que hasta el día de hoy, se desconoce).

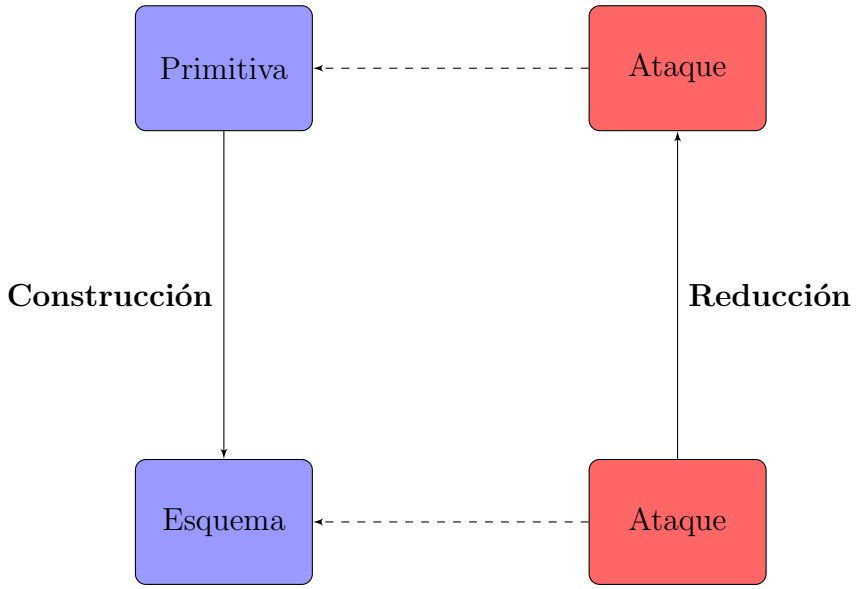


Figura 1.1: Construcción de pruebas

1.1. Criptografía asimétrica

En esta sección se introducirán algunos conceptos fundamentales de criptografía asimétrica, ya que serán de utilidad a la hora de explicar la siguiente sección (1.2) y algunos ejemplos posteriores.

La criptografía **asimétrica** o **de clave pública** engloba todos aquellos algoritmos criptográficos que usan dos funciones, $\mathcal{E}$ y $\mathcal{D}$, para cifrar y descifrar la información respectivamente. Intuitivamente, cifrar una información consiste en modificarla hasta hacerla ininteligible. El comportamiento de cada una de estas funciones está parametrizado por una **clave** distinta (para que el resultado no sea siempre el mismo para cada entrada), por lo que también es necesario un procedimiento de generación de claves $\mathcal{KG}$ para conseguir un par de claves pública-privada (pk, sk). Las funciones de cifrado y descifrado tienen la siguiente forma:

$$\mathcal{E}(pk, T) = C$$
$$\mathcal{D}(sk, C) = T$$

Donde T significa **texto en claro** o **mensaje**, y C, **texto cifrado** o **cifra**. Para que un esquema de criptografía asimétrica sea útil, siempre que el par (pk, sk) se haya generado en una sola llamada a $\mathcal{KG}$ se debe cumplir la siguiente igualdad:

$$\forall t \in T.\mathcal{D}(sk, \mathcal{E}(pk, t)) = t$$

Dicho de otra forma, la función de descifrado **invierte** la función de cifrado. $\mathcal{E}(pk)$ es lo que se denomina **función trampa** ya que su inversa, $\mathcal{D}(sk)$, es muy sencilla de computar si se conoce sk pero extremadamente difícil en caso contrario.

1.2. Secuencias de juegos

Hoy en día, las demostraciones relacionadas con criptografía siguen una estructura denominada **secuencia de juegos** [2]. Básicamente consisten en definir **juegos** entre un **desafiador** (*challenger*) y un **adversario** (*adversary*), ambos representados como programas probabilísticos:

Juego 1.1: IND-CPA ([3])

$(pk, sk) \leftarrow \mathcal{KG}()$;
$(m_0, m_1) \leftarrow A_1(pk)$;
$b \xleftarrow{\$} \{0, 1\}$;
$c \leftarrow \mathcal{E}(pk, m_b)$;
$\tilde{b} \leftarrow A_2(c)$

El Juego 1 define la propiedad de seguridad IND-CPA («Indistinguishability under Chosen Plaintext Attack») de los esquemas de cifrado asimétrico. El adversario está representado como la pareja de procedimientos (A_1, A_2). El desafiador crea un par

de claves (pk, sk), permite que el adversario elija dos textos (m_0, m_1) conociendo pk, cifra aleatoriamente uno de los dos y se lo entrega al adversario para que adivine cuál de los dos textos originales fue cifrado. Si el adversario no es capaz de acertar con una probabilidad mayor que 1/2 (intento a ciegas), se considera que el criptosistema cumple la propiedad IND-CPA.

En general, a cada juego se le asocia un **evento** S y una **probabilidad objetivo**, y se dice que cumple su propiedad de seguridad cuando para cualquier posible adversario, la posibilidad de que ocurra el evento S es arbitrariamente cercana a la probabilidad objetivo. En el Juego 1, el evento al finalizar la ejecución del juego sería $b = \tilde{b}$ (el adversario logra adivinar cuál de sus dos textos fue cifrado); y la probabilidad objetivo, 1/2. Una secuencia de juegos es un conjunto de juegos en el que la probabilidad de que suceda el evento S en cada juego es muy cercana a la del juego previo.

En [2] se demuestra que el criptosistema de clave pública ElGamal efectivamente cumple la propiedad IND-CPA realizando una reducción del juego original de IND-CPA (para $\mathcal{KG}$ y $\mathcal{E}$ los algoritmos de generación de claves y de cifrado, respectivamente, de ElGamal) a uno en el que se resuelve el problema de decisión de Diffie-Hellman, ambos con una probabilidad de S muy cercana. Esta demostración es válida porque el problema de decisión de Diffie-Hellman se considera inabordable a día de hoy.

1.3. EasyCrypt

El proceso de elaborar estas demostraciones es una tarea ardua y propensa a error, por lo que últimamente se han empezado a desarrollar herramientas para abordar la construcción y verificación de estas demostraciones en forma de secuencias de juegos, como CertiCrypt y su versión mejorada EasyCrypt [4].

EasyCrypt es un marco formado por un **lenguaje de programación** junto a un motor de resolución de **lógica de Hoare Relacional Probabilística** (ver sección 1.3.2) [3]. EasyCrypt es un proyecto de software libre, distribuido bajo los términos de la licencia CeCILL-B, y su código fuente está accesible desde la página del proyecto[1].

EasyCrypt funciona como un intérprete de un lenguaje de propósito general y de forma interactiva: cuando se le proporciona un fichero fuente a evaluar, lo recorre de arriba hacia abajo comprobando que todos los pasos de las demostraciones son correctos. Usando Proof General, un modo del editor de texto Emacs, es posible editar un fichero fuente mientras el intérprete lo va evaluando sobre la marcha, ayudando y guiando al usuario a medida que éste desarrolla cada demostración.

[1] `https://www.easycrypt.info`

EasyCrypt usa varios lenguajes de programación distintos a la hora de evaluar un código fuente. A continuación veremos cuáles son, con algunos ejemplos.

1.3.1. Lenguajes de especificacion

Estos lenguajes se usan para declarar y definir tipos, funciones, axiomas, juegos, oráculos, adversarios, entidades involucradas en las pruebas, etc.

Lenguaje de expresiones

El lenguaje principal de especificación de EasyCrypt es el de las expresiones, que define una serie de **tipos** (conjuntos de valores) y **operadores** sobre ellos. En EasyCrypt, siguiendo la notación del cálculo lambda tipado [5], la relación de tipado se representa con los dos puntos. Así, (b:*bool*) denota la pertenencia del valor b al tipo *bool*. EasyCrypt posee un potente sistema de tipos que soporta polimorfismo y tipos de orden superior. Los tipos de orden superior están parametrizados por otros tipos que se colocan delante: «*int list*» significa "lista de enteros". Los tipos se pueden generalizar usando una variable de tipo (variable precedida de una comilla simple): «'*a list*» significa "lista de cualquier tipo".

Los **operadores** son funciones definidas sobre tipos, y se introducen con la palabra reservada «**op**»: (**op** par : nat −> *bool*). La aplicación de un operador a un valor se realiza poniendo un espacio entre el operador y su argumento: (par 3). Aunque los operadores se definen de forma abstracta (sin proporcionar una implementación), la semántica de los operadores viene de la definición de lemas y axiomas que describen su comportamiento de forma observacional, o extensional. EasyCrypt asume que todos los tipos están habitados, por lo que los operadores son funciones totales. Para soportar operadores que reciban varios argumentos se usa la técnica de la **currificación** (*currying*), que convierte una función de múltiples argumentos en una que al ser aplicada al primer argumento devuelve una nueva función que recibe el segundo argumento, y así sucesivamente. Por ejemplo $f : (A \times B \times C) \to D$ pasaría a ser $f : A \to (B \to (C \to D))$, o lo que es lo mismo, $f : A \to B \to C \to D$.

En este ejemplo se ilustra la metodología general de definición de tipos y operadores en el lenguaje de expresiones de EasyCrypt:

Listado de código 1.1: Lenguaje de expresiones

```
type 'a predicate = 'a -> bool.
datatype 'a option = None | Some of 'a.
op find : 'a predicate -> 'a list -> 'a option.
axiom find_head :
  forall (x : bool) (l : bool list),
    hd l = x => find (lambda e, e = x) l = Some x.
```

Las primeras dos líneas defininen los tipos predicate y option. El primero representa funciones que devuelven *bool*, y el segundo, datos que pueden tomar los valores None o Some x (se usan para representar a nivel de tipos un dato que podría no existir).

A continuación se define el operador find, que dado un predicado y una lista de elementos, devuelve el elemento de la lista que satisface el predicado (o None si ninguno lo satisface).

Por último, se define un axioma llamado find_head: si dado un elemento x y una lista, siendo x el primer elemento de la lista, se llama a find con el predicado "igual a x", el resultado deberá ser (Some x).

En EasyCrypt hay dos formas de definir hechos: usando **axiomas** o **lemas**. La diferencia es que un axioma es válido por sí mismo, mientras que un lema se debe acompañar de una demostración, como se verá en el apartado 1.3.2.

Lenguaje de expresiones probabilísticas

Para poder razonar sobre distribuciones de probabilidad, EasyCrypt proporciona un tipo ('*a* distr) que representa sub-distribuciones discretas sobre '*a*. La principal herramienta para trabajar sobre sub-distribuciones es el operador
(**op** mu : '*a* distr −> ('*a* −> *bool*) −> *real*), que devuelve la probabilidad de un evento. Por ejemplo, la distribución uniforme sobre booleanos está definida en la biblioteca estándar de EasyCrypt de la siguiente manera (**charfun** es la función característica de p evaluado en x, es decir, devuelve 1 si p x = true y 0 si p x = false):

Listado de código 1.2: Distribución uniforme sobre bool

```
op dbool : bool distr.
axiom mu_def : forall (p : bool -> bool),
  mu dbool p =
    (1/2) * charfun p true +
    (1/2) * charfun p false.
```

Lenguaje pWhile

Los lenguajes de expresiones a menudo son inadecuados para definir juegos y otras estructuras de datos como esquemas de cifrado u oráculos, debido a la necesidad de representar algoritmos secuenciales, entidades con estado, etc. Para ello, EasyCrypt implementa un lenguaje distinto llamado pWhile (probabilistic While) [3].

Gramática 1.1: Lenguaje pWhile

$$
\begin{aligned}
\mathcal{I} ::= & \ \mathcal{V} \leftarrow \mathcal{E} \\
| & \ \mathcal{V} \xleftarrow{\$} \mathcal{D}\mathcal{E} \\
| & \ \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C} \textbf{ else } \mathcal{C} \\
| & \ \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \\
| & \ \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, ..., \mathcal{E}) \\
\mathcal{C} ::= & \ \textbf{skip} \\
| & \ \mathcal{I}; \ \mathcal{C}
\end{aligned}
$$

1.3.2. Lenguajes de demostración

Estos lenguajes se usan para definir lemas y demostrarlos.

Juicios

Los juicios son proposiciones que se pueden realizar en EasyCrypt, y tienen asociado un valor de verdad (pueden ser ciertos o falsos). Para demostrar su veracidad, un juicio deberá demostrarse usando el lenguaje de tácticas (detallado a continuación). Además de los juicios expresados en lógica de primer orden, EasyCrypt cuenta con diversos tipos de juicio derivados de la Lógica de Hoare que se usan para razonar sobre la ejecución de programas:

- Lógica de Hoare (*HL*). Tienen la siguiente forma:

$$c : P \Longrightarrow Q$$

  donde P y Q son aserciones (predicados) y c es un mandato o programa. P es la **precondición** y Q es la **postcondición**. El juicio expresa que si la precondición se cumple al momento de ejecutar el mandato, la postcondición también se cumplirá cuando éste termine (si es que lo hace).

- Lógica de Hoare probabilística (*pHL*). A los juicios de Hoare anteriores se les asigna una probabilidad, que puede ser exacta o una cota superior/inferior.

$$[c : P \Longrightarrow Q] \leq \delta$$
$$[c : P \Longrightarrow Q] = \delta$$
$$[c : P \Longrightarrow Q] \geq \delta$$

- Lógica de Hoare relacional probabilística (*pRHL*). Tienen la siguiente forma:

$$c_1 \sim c_2 : \Psi \Longrightarrow \Phi$$

  En este caso, el juicio expresa que si la precondición Ψ se cumple antes de la ejecución de c_1 y c_2, tras su ejecución también se cumplirá la postcondición Φ. En este caso las (pre,post)condiciones son **relaciones** entre las memorias de los dos programas, por lo que este tipo de juicios expresa una **equivalencia** entre dos programas con respecto a ellas. Además, en este caso los programas c_1 y c_2 son probabilísticos: su evaluación produce como resultado una distribución de probabilidad sobre los posibles estados en los que se podrá encontrar la memoria al ser ejecutado.

Tácticas

Para demostrar juicios, EasyCrypt cuenta con un lenguaje basado en tácticas (*tactics*). Una demostración consiste en una secuencia de tácticas que se van ejecutando iterativamente, donde cada táctica aplica una transformación sintáctica al **objetivo** actual (la fórmula que se está demostrando) hasta que el último resuelve finalmente el objetivo.

Listado de código 1.3: Uso de tácticas

```
lemma hd_tl_cons :
  forall (xs:'a list),
    xs <> [] => (hd xs)::(tl xs) = xs.
proof.
  intros xs.
  elim / list_ind xs.
  simplify.
  intros x xs' IH h {h}.
  rewrite hd_cons.
  rewrite tl_cons.
  reflexivity.
qed.
```

En este ejemplo obtenido de la biblioteca estándar de EasyCrypt (ligeramente modificado para mejorar su comprensión) se ilustra cómo se realizan las demostraciones. Lo que aparece a continuación del nombre del lema es su especificación formal en lógica de primer orden: una propiedad acerca de la descomposición de las listas en cabeza (*head*) y cola (*tail*). Lo que aparece a continuación es la demostración del lema, con una táctica por cada línea. La demostración se vale de algunos axiomas definidos anteriormente como el de la inducción sobre listas (list_ind) o la definición de «hd» y «tl» (hd_cons, tl_cons).

Una táctica que vale la pena destacar es smt, que intenta resolver el objetivo actual de la demostración invocando **resolvedores SMT** («Satisfiability Modulo Theories») (ver sección 2.3). Estas herramientas están diseñadas para intentar encontrar soluciones a fórmulas expresadas en lógica de primer orden, permitiendo ciertas interpretaciones adicionales sobre los símbolos. Normalmente soportan ciertas teorías muy útiles en criptografía como arrays, listas, aritmética entera, etc., y a menudo son capaces de resolver de forma automática partes repetitivas y tediosas de las demostraciones, liberando de esa carga de trabajo al criptógrafo.

1.4. Objetivos

Para este trabajo se ha decidido contribuir dentro de lo posible a mejorar el sistema EasyCrypt, desarrollado principalmente en el Instituto IMDEA Software en Madrid. Aunque es un sistema que ya se ha utilizado para demostrar la seguridad de construcciones criptográficas ampliamente usadas como el criptosistema de Cramer-Shoup o el modo de operación CBC para cifradores de bloque, EasyCrypt sigue en constante desarrollo y tiene muchas posibilidades de ampliación. En concreto, en este trabajo se abordarán e intentarán resolver dos problemas de distinta índole:

- **La posibilidad de trabajar con el coste de algoritmos.** Para poder deducir la capacidad computacional requerida para romper un criptosistema se necesita un método para especificar el coste computacional asociado a computar ciertas operaciones. El objetivo es ser capaz de relacionar la seguridad de un sistema con los recursos del atacante, por ejemplo para afirmar que el sistema es seguro mientras los recursos del atacante no superen cierto límite.

  Trabajar con costes implica, por una parte, **especificarlos**, y por otra **usarlos para demostrar** lemas. La primera parte del proyecto consistirá en la implementación de un mecanismo que permita su **especificación**, ya que implementar un mecanismo para demostrar con ellos es una tarea considerablemente difícil y que supera los objetivos del Trabajo de Fin de Grado.

- **Mejorar la usabilidad del sistema.** Idealmente, EasyCrypt debería ser utilizado por criptólogos como herramienta de verificación, pero para ello es necesario hacer su uso lo más sencillo posible. Aunque EasyCrypt es bastante más usable que su predecesor CertiCrypt, sigue estando lejos de ser sencillo. La primera barrera de entrada consiste en su propia instalación: es necesario clonar el repositorio donde está alojado el código y compilarlo junto con sus dependencias (proceso que puede llegar a tardar hasta una hora). Además, toda la interacción con EasyCrypt se realiza a través de ProofGeneral y Emacs, por lo que tiene que estar instalado y el usuario debe de tener cierta experiencia para usarlo cómodamente.

Una forma relativamente sencilla de eliminar esta barrera de entrada es ofrecer una interfaz web con todo lo necesario para usar EasyCrypt sin tenerlo instalado en el sistema local. Además de ahorrar tiempo en la instalación y en aprender a usar Emacs, haría más sencillo trabajar desde múltiples sitios al tener toda la estructura del proyecto centralizada en el servidor remoto.

En la segunda parte del proyecto se llevará a cabo el diseño e implementación de este sitio web, con el fin de hacer más sencillo usar EasyCrypt y fomentar su utilización en los círculos de gente que se dedica a la criptografía.

Capítulo 2

DESARROLLO: COSTE

La principal aportación de EasyCrypt con respecto a otros demostradores de teoremas como Coq[1] o Isabelle/HOL[2] son todas las facilidades que proporciona para trabajar, concretamente, en el ámbito de la criptografía. Algunos ejemplos son el soporte a demostraciones basadas en secuencias de juegos, la riqueza de juicios (lógica de Hoare y variantes) o su exhaustiva biblioteca estándar, que define tipos de datos habituales (cadenas de bits, distribuciones probabilísticas, etc.), funciones para manipularlos, axiomas y lemas de uso común, etc.

Un concepto muy significativo en criptografía es el del **coste**. El coste de un algoritmo es una medida de los recursos (tiempo, espacio, ...) que requiere su ejecución. Como hemos visto en la introducción, salvo en casos muy concretos (OTP [6]), la seguridad en criptografía depende de la capacidad computacional del adversario. Por tanto, existen muchas propiedades acerca de la seguridad de un criptosistema que de una u otra forma dependen del coste de llevar a cabo un ataque contra él.

A lo largo de este capítulo se detallará el proceso que se ha seguido para implementar la funcionalidad que permita a la especificación del coste de algoritmos en EasyCrypt.

[1] `http://coq.inria.fr/`
[2] `http://isabelle.in.tum.de/`

2.1. Motivación

Supongamos que el usuario está construyendo una demostración de que un criptosistema de clave pública es seguro siempre que los recursos del adversario no sean superiores a cierto límite. En este caso, por "seguro" nos referimos a que el adversario no es capaz de descifrar un texto cifrado sin poseer la clave privada correspondiente. Como hemos visto en la sección 1.1, esto es equivalente a decir que el adversario no es capaz de **invertir** la función de cifrado.

La primera parte de la prueba consistiría en demostrar que $\mathcal{E}(pk)$ y $\mathcal{D}(sk)$ realmente constituyen una función trampa. En este caso, la seguridad dependería única y exclusivamente de la dificultad de adivinar sk. Suponiendo que el sistema no tenga fallos que permitan extraer información de alguna otra forma, el adversario no tendrá más remedio que usar la **fuerza bruta**: probar todas las claves posibles hasta encontrar la correcta. Por este mismo motivo, la complejidad de la clave (longitud, aleatoriedad) es crucial: adivinar una clave aleatoria de 32 bits requerirá como mucho $2^{32} = 4294967296$ intentos, mientras que una clave aleatoria de 256 bits requerirá una cantidad de intentos del orden de la cantidad de átomos en el universo.

En este punto, el criptosistema será seguro siempre que se pueda probar que el adversario no es capaz de adivinar la clave correcta en un tiempo razonable. Por ello, si queremos proporcionar una cota exacta de la cantidad de trabajo necesario que requiere romper el criptosistema (coste) necesitamos una forma de especificar los costes de cada operación, combinarlos para obtener el coste del algoritmo completo, y tenerlos en cuenta a la hora de realizar las demostraciones. A lo largo de este capítulo veremos cómo se ha implementado esta funcionalidad.

2.2. Diseño

2.2.1. Operador especial de coste

Para empezar, necesitamos una forma de hacer referencia al coste asociado a un operador. Una primera aproximación podría ser usar una **función** que dado un operador de cualquier tipo (constant o función) devuelva su coste en forma de número entero positivo:

Listado de código 2.4: Definición 1 del coste

```
op cost : 'a -> int.
```

Sin embargo, hay un problema con esta solución. En EasyCrypt la igualdad se define de forma extensional, por lo que dos funciones se consideran la misma cuando su resultado es el mismo para las mismas entradas. Dicho de un modo más formal:

$$\forall f, g.(\forall x.f(x) = g(x)) \Rightarrow f = g$$

Esta aproximación nos permitiría deducir que dos funciones extensionalmente iguales tienen siempre el mismo coste, ya que $f = g$ implica $cost(f) = cost(g)$. El problema es que dos funciones extensionalmente iguales no tienen por qué tener el mismo coste, ni siquiera la misma clase de complejidad (por ejemplo, cualquier algoritmo que se ejecute en tiempo lineal se puede reducir a otro algoritmo que se ejecute en tiempo cuadrático).

Una segunda aproximación es la de usar un **operador especial** definido dentro del propio compilador (en lugar de en una biblioteca del lenguaje, como hubiera sido el caso anterior) para evitar que se le apliquen las mismas suposiciones que al resto de funciones y así evitar inconsistencias en la lógica:

Listado de código 2.5: Definición 2 del coste

```
cost[myop, arg1, arg2, ...] : int
```

El operador especial **cost** devolvería el coste en forma de entero positivo, como en el caso anterior, pero esta vez sería necesario especificar la totalidad de los argumentos del operador cuyo coste estemos definiendo. El operador debe de estar declarado anteriormente, y los argumentos pueden ser valores literales, otros operadores o variables cuantificadas universalmente (como queremos diferenciar entre la cuantificación universal introducida por la construcción *forall* y ésta, tendremos que indroducir un ámbito especial con el axioma de coste, explicado a continuación).

2.2.2. Axioma de coste

Además de poder hacer referencia a los costes con el operador especial de coste, necesitamos una forma de poder definirlos. Para esto necesitamos otra construcción nueva del lenguaje: el **axioma de coste**:

Listado de código 2.6: Axioma de coste

```
caxiom name param1 param2 ... : spec.
```

Como se puede apreciar, es muy similar a la definición de un axioma normal en EasyCrypt (ver listado 1.1). La diferencia son los parámetros que van a continuación del nombre del axioma. Estas variables se ligan implícitamente con un cuantificador universal para ser referenciadas dentro del operador especial de coste, que puede aparecer dentro de la especificación.

En este ejemplo se axiomatiza el hecho de que ejecutar la función sum tiene coste cero cuando uno de sus argumentos es cero:

Listado de código 2.7: Coste (parcial) de sum

```
caxiom sum_zero x :
  cost[sum, 0, x] = 0 &&
  cost[sum, x, 0] = 0.
```

En este otro ejemplo se define el coste de ejecutar la función find (listado 1.1), que busca en una lista el elemento que satisface un predicado:

Listado de código 2.8: Coste de find

```
caxiom find_cost (l : 'a list) (p : 'a predicate) (x : 'a) :
  cost[find, p, l] <= cost[p, x] * length l.
```

Una vez definida la funcionalidad que se quiere conseguir (**operador especial de coste** y **axioma de coste**), pasamos a analizar el proceso de su implementación.

2.3. Arquitectura de EasyCrypt

EasyCrypt está escrito en el lenguaje de programación OCaml[1], que forma parte de la familia ML de lenguajes especialmente diseñados para el desarrollo de demostraciones. La unidad de compilación básica de OCaml es el **módulo**, consistente en un fichero de implementación con extensión **.ml** y un fichero de interfaz con extensión **.mli** donde se declaran las funciones, tipos, etc. La compilación es relativamente compleja y se automatiza con ayuda de GNU Make[2] y ocamlbuild[3].

EasyCrypt usa varios programas externos para llevar a cabo ciertas tareas, pero los más destacables son los dos siguientes:

- **Resolvedores SMT**. Existen multitud de programas que deciden la satisfacibilidad de instancias SMT. Como ya se dijo anteriormente, una instancia de fórmula SMT está expresada en lógica de primer orden y tiene interpretaciones en teorías como los números reales, vectores, bits, etc. Durante una demostración en EasyCrypt se pueden invocar estos resolvedores para tratar de encontrar una solución que satisfaga el objetivo actual usando la táctica smt. Internamente, EasyCrypt estará comunicándose con uno o varios resolvedores y obteniendo la respuesa al problema, si es que existe. Para contar con una única interfaz de programación ante la diversidad de resolvedores, EasyCrypt usa la plataforma **Why3**[4].

[1] http://ocaml.org/
[2] http://www.gnu.org/software/make/
[3] http://caml.inria.fr/pub/docs/manual-ocaml-400/manual032.html
[4] http://why3.lri.fr/

- **Menhir**[1]. Menhir es una biblioteca para generar analizadores léxicos y sintácticos en OCaml. Se verá con mayor profundidad en las secciones 2.4 y 2.5, tras explicar las fases del análisis del lenguaje.

A fin de cuentas, EasyCrypt es un intérprete: lee un fichero fuente y realiza acciones en función de su contenido. Por ello, el componente fundamental es el **analizador del lenguaje**, cuya tarea es transformar el código fuente (secuencia de caracteres) en un árbol sintáctico abstracto anotado con información de tipos, o mostrar un error y finalizar en caso de que el programa esté mal formado.

El análisis del lenguaje consta de varias fases encargadas de abordar niveles de abstracción distintos. Conceptualmente, las fases de análisis son las siguientes:

- **Análisis léxico** Esta fase es la primera que se lleva a cabo y la que tiene un menor nivel de abstracción; su entrada es el propio fichero fuente. Durante el análisis léxico se transforma la secuencia de caracteres que forma el código fuente en unidades mínimas de información (**tokens**): número, cadena de caracteres, identificador, comentario, inicio de bloque, etc. La salida de esta fase es una lista conteniendo todos los tokens que describen el programa original, formando una estructura de datos muy simple pero más sencilla de recorrer y manejar.

- **Análisis sintáctico** Durante esta fase se transforma la lista de tokens generada durante el análisis léxico en el denominado **árbol sintáctico abstracto** (AST), una estructura de datos que contiene la estructura sintáctica del código fuente. Este analizador está especificado por la gramática formal del lenguaje (gramáticas libres de contexto, por lo general), cuyas reglas determinan el algoritmo a seguir durante la construcción del árbol sintáctico.

- **Análisis semántico/contextual** Durante esta fase se recorre el árbol sintáctico generado previamente y se anota con información de tipos, se construye la tabla de símbolos y se comprueba que el programa, además de estar bien formado sintácticamente, tiene sentido semánticamente.

EasyCrypt implementa las dos primeras etapas en los módulos EcLexer y EcParser, define el árbol sintáctico abstracto en EcParseTree, y la generación del árbol anotado con tipos es responsabilidad de módulos especializados en su ámbito. En nuestro caso la transformación semántica la realiza el módulo EcTyping, con las estructuras de datos anotadas definidas en EcFol, ya que son los módulos encargado de gestionar fórmulas de primer orden, como veremos con más detalle en las secciones 2.5 y 2.6. Los módulos EcLexer y EcParser hacen uso de la biblioteca Menhir mencionada anteriormente para generar los analizadores. Estos módulos contienen las definiciones

[1] `http://gallium.inria.fr/~fpottier/menhir/`

de tokens y las reglas de las gramáticas regular y libre de contexto usadas para indicar, respectivamente, el comportamiento de los analizadores léxico y sintáctico. Todo este proceso aparece ilustrado en la figura 2.1.

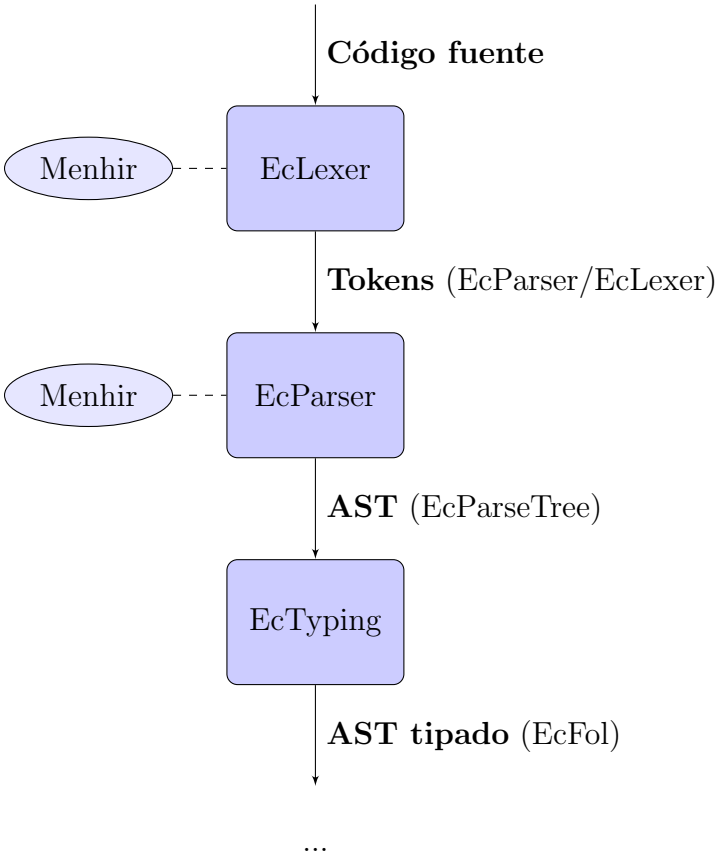


Figura 2.1: Analizador del lenguaje de EasyCrypt

Para la implementación del coste es necesario modificar cada uno de los módulos involucrados en este proceso. A continuación analizaremos las etapas de la implementación tanto del operador especial de coste como del axioma de coste.

2.4. Modificaciones al analizador léxico

EcLexer es el módulo que define el analizador léxico. Este módulo está contenido en el fichero «ecLexer.mll», cuya extensión refleja el hecho de que no es un fichero OCaml válido, sino un fichero de definición de analizadores léxicos que deberá ser procesado por Menhir para generar el analizador real.

La función principal de este módulo es asociar expresiones regulares con sus tokens correspondientes. Por ejemplo, la expresión regular **[0-9]*** emite el token UINT (Unsigned INTeger) parametrizado por el valor de la cadena coincidente convertida a número entero.

La única modificación que debemos realizar a este módulo es el reconocimiento de nuestros dos nuevos tokens: **cost** y **caxiom**. Estos tokens son palabras reservadas: identificadores con significado especial dentro del lenguaje. En EcLexer, cuando se encuentra un identificador, se busca dentro de un **hash map** que relaciona palabras reservadas con sus respectivos tokens: en caso de existir la clave, se emite el token asociado; en caso contrario, se emite el token genérico IDENT. Por tanto, basta con añadir dos entradas a este hash map para que EcLexer emita los tokens COST y CAXIOM cuando encuentre los identificadores ‘cost‘ y ‘caxiom‘, respectivamente:

```
let _keywords = [
  "admit" , ADMIT;
  "forall", FORALL;
  "exists", EXIST;
  "pragma", PRAGMA;
  (* ... *)

  "cost"  , COST;
  "caxiom", CAXIOM
]

let keywords = Hastbl.create 100
let _ = List.iter (fun x, y -> Hashtbl.add keywords x y) _keywords
```

Listado de código 2.1: ecLexer.mll

2.5. Modificaciones al analizador sintáctico

Una vez el analizador léxico reconoce las nuevas palabras reservadas que hemos introducido, es tarea del analizador sintáctico reconocer las estructuras que forman los tokens. El analizador sintáctico de EasyCrypt está implementado en el módulo EcParser que, al igual que el analizador léxico, no se define usando código OCaml normal: el fichero que lo contiene es «ecParser.mly», y debe ser procesado por Menhir para generar el auténtico analizador sintáctico.

La función de este módulo es la especificación de la gramática de EasyCrypt como tal. Esto se realiza describiendo las reglas en formato BNF «Backus-Naur Form». La sintaxis concreta que usa Menhir es la siguiente:

```
<nombre_regla>:
  | [<tokens o reglas>] { <valor_semántico> }
  | [<tokens o reglas>] { <valor_semántico> }
  | ...
  ;
```

Cada regla define un símbolo no terminal (<nombre_regla>) que la identifica, así como uno o varios **grupos de producción** que representan las posibles expansiones de la regla. Cada grupo de producción comienza por el símbolo de la barra vertical «|», continúa con una serie de símbolos que pueden hacer referencia a tokens o a otras reglas, y termina con una expresión entre llaves «{ , }» conteniendo las **acciones semánticas** de la regla (los efectos secundarios que se llevan a cabo durante su expansión).

Como el objetivo de esta fase es generar un árbol sintáctico abstracto, cada regla será la responsable de generar un nodo con su representación abstracto como acción semántica. Por eso, antes de escribir nuestras reglas modificaremos el fichero que define el árbol sintáctico.

Para empezar, añadiremos un nodo que represente el operador especial de coste. De ahora en adelante lo llamaremos **expresión** de coste, ya que al fin y al cabo es una expresión (se evalúa a un número entero) y no provoca ambigüedad con el resto de operadores normales. La definición es la siguiente:

```
type pcexpr = (pqsymbol * pexpr list) located
```

Listado de código 2.2: Nodo: expresión de coste (ecParseTree.ml)

Es decir, que "pcexpr" («parsed cost expression») es un tipo de datos cuyos valores son tuplas «(símbolo, [expresiones])» que almacenan respectivamente el operador cuyo coste se está evaluando y sus argumentos, y además contienen información de su ubicación en el código fuente: fila y columna («located»).

A continuación definiremos el nodo correspondiente al axioma de coste:

```
type pcaxiom = {
  pca_name  : psymbol;
  pca_cargs : ptybindings option;
  pca_spec  : pformula;
}
```

Listado de código 2.3: Nodo: axioma de coste (ecParseTree.ml)

El axioma de coste es un "record" (estructura de datos con varios campos) que almacena su nombre, argumentos (opcionales) y especificación. La especificación es una fórmula de primer orden: expresiones que pueden estar cuantificadas existencial o universalmente. Por tanto, también necesitaremos modificar el tipo de datos fórmula para que pueda contener expresiones de coste:

```
type pformula  = pformula_r located

and pformula_r =
  | PFhole
  | PFint    of int
  | PFtuple  of pformula list

  (* ... *)

  | PFcost   of pcexpr
```

Listado de código 2.4: Nodo: fórmulas (ecParseTree.ml)

Una vez tenemos los nodos definidos, volvemos al analizador sintáctico para definir las reglas:

```
cexpr:
  | COST '[' op=qident ']'
      { (* ... *)
        (op, []) }
  | COST '[' op=qident ',' ps=plist0(expr, ',') ']'
      { (* ... *)
        (op, ps) }
  ;

caxiom:
  | CAXIOM x=ident args=ptybindings? ':' spec=form
      { { pca_name=x; pca_cargs=args; pca_spec=spec } }
  ;
```

Se puede observar cómo cada regla, además de definir la sintaxis, crea su nodo correspondiente y lo devuelve como valor semántico. Esto es todo lo que necesitamos para que EasyCrypt reconozca sintácticamente las nuevas construcciones.

A continuación pasaremos a la última etapa de la implementación del coste, y la más compleja: modificar el analizador semántico.

2.6. Modificaciones al analizador semántico

El análisis semántico en EasyCrypt fundamentalmente consiste en la comprobación de tipos (incluyendo de ámbito y entorno). A continuación se expondrá a muy alto nivel cómo se ha abordado la implementación, ya que en EasyCrypt el análisis contextual es muy exhaustivo y complejo. Para más detalles se aconseja al lector interesado que obtenga el código fuente desde la página web del proyecto.

El módulo principal encargado de implementar la funcionalidad del analizador semántico (comprobar los tipos y anotar el árbol sintáctico abstracto) es «EcTyping». Este módulo exporta funciones que transforman fórmulas definidas en EcParseTree, sin información semántica, en fórmulas definidas en el módulo EcFol, obtenidas tras realizar las comprobaciones necesarias y portando información semántica.

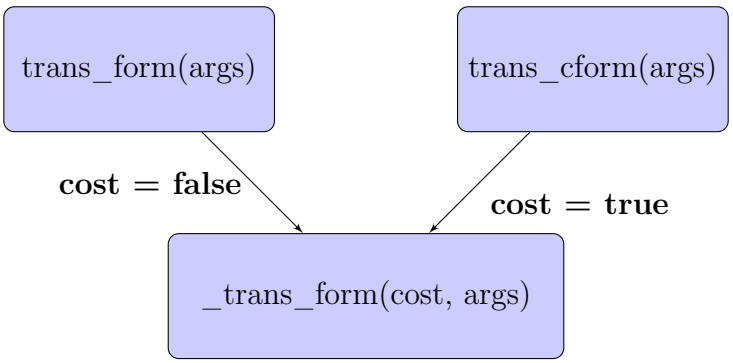
La función que más nos interesa es trans_form:

```
val trans_form : env -> unienv -> pformula -> ty -> EcFol.form
```

Listado de código 2.5: Tipo de la función trans_form (ecTyping.mli)

Sin entrar en detalles, esta función convierte una fórmula («pformula») en una fórmula anotada («EcFol.form») en función del entorno léxico («env»), el entorno de unificación («unienv») y el tipo de datos esperado («ty»). Necesitamos una variante de esta función que soporte también fórmulas que contengan expresiones de coste, pero que solo se pueda invocar cuando la fórmula aparezca dentro de un axioma de coste.

Para ello, modificaremos la función original para que reciba un argumento booleano adicional "cost" que si esa fórmula puede contener expresiones de coste o no, y para no romper la compatibilidad, la convertiremos en una función auxiliar «_trans_form» y crearemos dos funciones «trans_form» y «trans_cform» que la invoquen con el valor correcto de "cost":



Además, habrá que añadir el caso en que «_trans_form» sea invocado con una expresión de coste:

```
  | PFcost pcexpr ->
      if cost
      then f_cost (transcexp env ue pcexpr)
      else tyerror f.pl_loc env NotInCostEnv
```

Listado de código 2.6: Transformación de cformula para expresiones de coste (ecTyping.ml)

Como podemos ver, lanza un error en caso de incluirse una expresión de coste fuera de un entorno válido (axioma de coste). El valor de retorno lo delega a la función transcexp, que busca el operador objetivo en el entorno, aplica recursivamente la transformación semántica sobre sus argumentos y unifica sus tipos (para que sea ilegal, por ejemplo, escribir **cost**[sum, true, 2]). Aquí concluye la transformación semántica de la expresión de coste.

La otra parte, el axioma de coste, es más sencilla de implementar. La función que lleva a cabo su transformación semántica ha de definirse en el módulo EcCommands, ya que el axioma de coste es una sentencia básica del lenguaje (es uno de los pasos que puede ejecutar el evaluador). Su trabajo es crear un nuevo entorno para las variables cuantificadas universalmente, transformar recursivamente su especificación bajo ese entorno ampliado y añadirlo al ámbito global para su posterior referencia.

Capítulo 3

DESARROLLO: INTERFAZ WEB

Como ya mencionamos en la sección 1.4, EasyCrypt es un programa complejo y una de las prioridades para favorecer su adopción es facilitar su uso en la medida de lo posible.

A lo largo de este capítulo se describirá el proceso de diseño e implementación de una interfaz web para trabajar con EasyCrypt remotamente desde un navegador.

3.1. Diseño

La mayor inspiración que tenemos para el diseño es el del propio Proof General, que es con lo que se trabaja en EasyCrypt normalmente (figura 3.1).

En nuestro caso, la parte fundamental de la interfaz será el editor de código fuente. Los resultados de la evaluación del código se mostrarán a la derecha del mismo, como en Proof General, y añadiremos un navegador de ficheros (ya que el código se almacena remotamente). Por último, necesitaremos una barra de navegación en la parte superior para realizar actividades complementarias como crear proyectos, iniciar/cerrar sesión, etc. En la figura 3.2 se puede ver el primer prototipo de la página principal de la web[1].

En cuanto al diseño del sistema en general, se han utilizado varios marcos y herramientas web para implementar toda la funcionalidad necesaria en un nivel superior, evitando en la medida de lo posible el uso directo de las tecnologías web básicas (HTML, CSS, JavaScript), más tediosas y propensas a error:

[1] A pesar de que la web debe contar con muchas otras páginas (pantalla de inicio, formulario de creación de cuentas de usuario, etc.), en este documento no se mencionarán por no ser tan relevantes en su interacción con EasyCrypt.

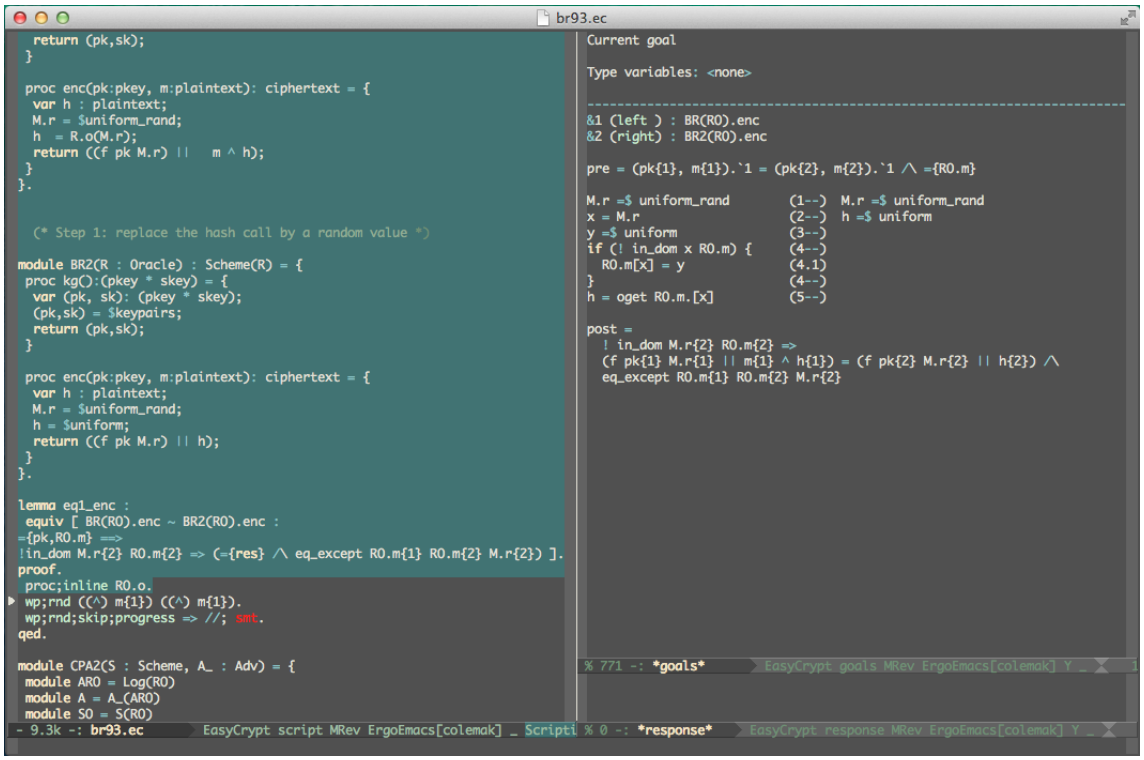


Figura 3.1: Desarrollo de pruebas en EasyCrypt usando Emacs y Proof General. A la izquierda: código fuente; a la derecha: objetivos y resultados.

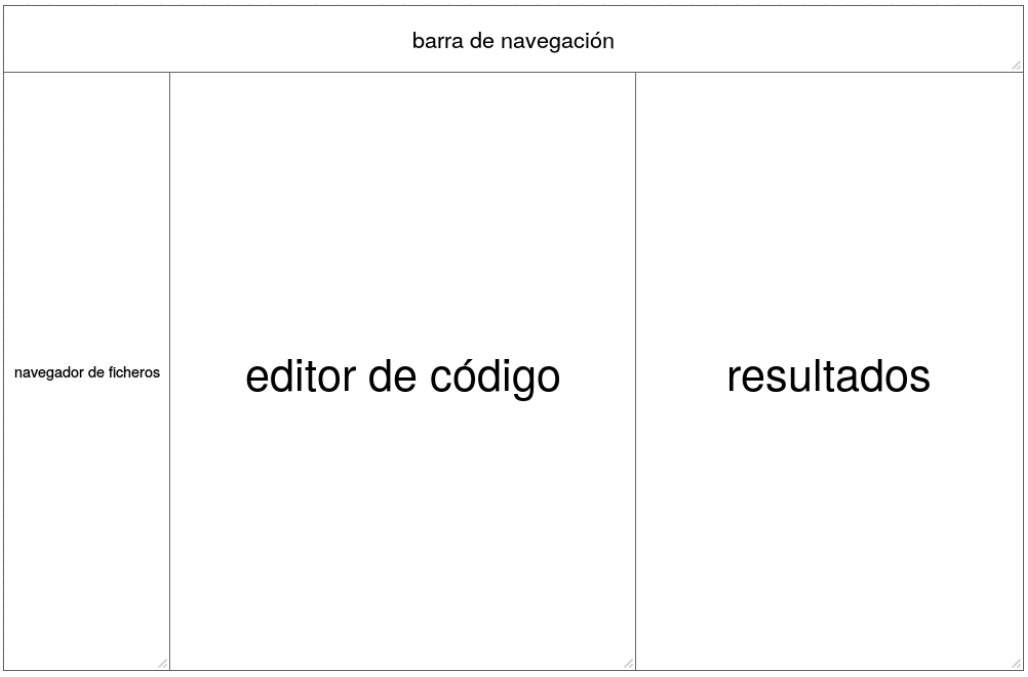


Figura 3.2: Diseño de la página principal de la interfaz web

- **Django**[1]. La base del sistema es Django, un marco web usado para implementar la funcionalidad en el lado del servidor. Django está escrito en Python, que es también el lenguaje que se usa para implementar el resto de la funcionalidad. Se ha utilizado el módulo crispy-forms[2] para poder definir los formularios HTML (inicio de sesión, creación de usuarios, etc.) directamente en Python.

- **Twitter Bootstrap**[3]. Bootstrap es un marco orientado al desarrollo de interfaces web. Sus características más interesantes son la gestión del esquema de la web usando un sistema de rejillas, una gran cantidad de clases CSS predefinidas y funcionalidad JavaScript como generación de ventanas «modales» (flotantes).

- **jQuery**[4]. En el lado del cliente hay mucha funcionalidad dinámica que debe ser implementada en JavaScript. Buena parte de este trabajo consiste en manipular la estructura HTML del documento: el DOM. Para este tipo de tarea se ha usado jQuery, que tiene una gran biblioteca de funciones para crear, eliminar, buscar y modificar nodos HTML, por ejemplo a la hora de mostrar el navegador de ficheros. Asimismo, para poder tener varios ficheros abiertos al mismo tiempo, se ha usado un widget de gestión de pestañas proporcionado por la biblioteca jQuery UI[5].

- **Ace**[6]. Ace es un editor de código fuente diseñado para integrarse en sitios web. Además, como permite que la misma instancia del editor pueda tener varias sesiones distintas (estados), podemos asociar una sesión a cada pestaña (fichero abierto) y guardar por separado sus estados: posición del cursor, código evaluado, etc.

- **WebSockets**[7]. Para evaluar remotamente el código definido en el editor es necesario disponer de una conexión persistente y bidireccional con un servidor que ejecute una instancia de EasyCrypt. Precisamente por estas necesidades se ha decidido implementar las comunicaciones entre el cliente web y el servidor de EasyCrypt usando WebSockets, una tecnología relativamente reciente pero soportada por prácticamente todos los navegadores actuales (a la fecha de redacción del presente documento, el único navegador que no soporta WebSockets es Opera Mini). Para la implementación de WebSockets en el lado del servidor de EasyCrypt se ha usado una biblioteca para Python llamada **Tornado**[8].

[1] `https://www.djangoproject.com/`
[2] `https://django-crispy-forms.readthedocs.org/en/latest/`
[3] `http://getbootstrap.com/`
[4] `http://jquery.com/`
[5] `http://jqueryui.com/`
[6] `http://ace.c9.io/`
[7] `http://www.websocket.org/`
[8] `http://www.tornadoweb.org//`

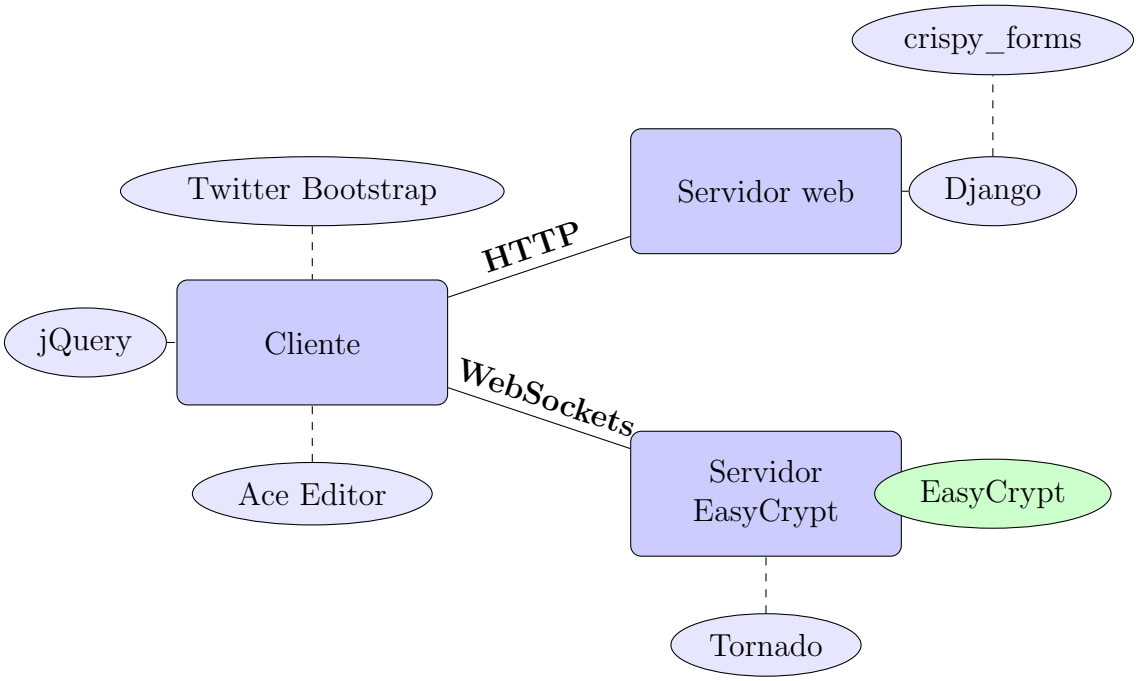


Figura 3.3: Arquitectura de la interfaz web

La figura 3.3 aporta una visión general del sistema completo. A continuación se describirá cada uno de los componentes que lo forman, la forma en que interactúa con el resto y el proceso de su implementación.

3.2. Implementación: servidor web

El servidor web es uno de los tres componentes fundamentales del sistema. A pesar de todo, su papel es por encima de todo el de almacenaje de información y punto de entrada al resto de los componentes. Concretamente, una vez el usuario haya iniciado sesión, su papel será el de servir el código que implementa el cliente y gestionar los datos que se almacenan remotamente. Tras esto, casi toda la interacción con el sistema involucrará exclusivamente al cliente (sección 3.3) y al servidor EasyCrypt (sección 3.4).

El servidor web está implementado en Python, soportado por el marco de desarrollo web Django y su gran colección de bibliosecas. Django sigue el patrón modelo-vista-controlador (MVC) a la hora de organizar el código, por lo que nos centraremos en cada una de estas partes por separado para tener una idea general de la estructura del servidor web.

3.2.1. Modelo

El modelo es el esquema de los datos que se van a manejar en la aplicación. En este caso va a ser muy simple, ya que los únicos datos que se van a manipular son **proyectos** y **ficheros**. Normalmente también necesitaríamos entidades como «User» para implementar la autenticación, pero Django proporciona esa funcionalidad por defecto.

Nuestras entidades serán, por tanto, las siguientes:

- Entidad: **Project**
  - Atributo: **name** (tipo: string)
  - Atributo: **owner** (tipo: User)
- Entidad: **File**
  - Atributo: **name** (tipo: string)
  - Atributo: **contents** (tipo: string)
  - Atributo: **project** (tipo: Project)

Es decir, cada usuario (User) posee uno o varios proyectos (Project), que a su vez contienen uno o varios ficheros (File).

En Django los modelos de datos se almacenan en un fichero especial «models.py». Cada entidad se representa como una clase de Python, y los atributos del modelo son atributos de instancia:

```
class Project(models.Model):
    name = models.CharField(max_length=200)
    owner = models.ForeignKey(get_user_model())
    # ...

class File(models.Model):
    name = models.CharField(max_length=200)
    contents = models.TextField()
    project = models.ForeignKey(Project)
    # ...
```

Listado de código 3.1: models.py

Una vez definidos los modelos, Django genera automáticamente la base de datos (modelo → tabla, atributo → columna, ...) y la reifica (representando cada entrada en la base de datos como un objeto en Python), para poder hacer un uso totalmente transparente de ella. Además, Django incluye varios controladores de gestores de bases de datos que abstraen su acceso y hacen posible intercambiarlas sin mayor problema. En nuestro caso, debido a la facilidad de uso hemos usado el gestor de bases de datos **SQLite**[1] durante el desarrollo, aunque debería ser trivial cambiarlo en el futuro por cualquier otro gestor en caso de darse la necesidad.

3.2.2. Vista

La vista es el subsistema encargado de generar y servir la parte visual de la web: ficheros HTML (que a menudo incluyen también código JavaScript y CSS).

En Django, la vista la implementa un sistema de plantillas («templates»)[2]. Una plantilla no es más que un fichero HTML que Django modifica en tiempo de ejecución según le indique la presencia de ciertas directivas. Por ejemplo, todo lo que aparece dentro de dobles llaves «{{, }}» en la plantilla se considera una expresión de Python y se sustituye por su valor en el momento de realizar la sustitución.

Las plantillas se almacenan normalmente en un directorio "/templates", separadas del resto del código del proyecto. Para desarrollar la interfaz web ha sido necesario escribir un total de 4 plantillas, descritas a continuación junto con una captura de pantalla de su resultado una vez servidas:

- **base.html**, donde se define la estructura HTML básica del sitio web y se cargan recursos externos como el tema CSS, bibliotecas de JavaScript, etc. No se usa directamente, sino que se importa desde otras plantillas usando la directiva **extends** de Django.

[1] `http://www.sqlite.org/`
[2] `https://docs.djangoproject.com/en/dev/topics/templates/`

- **login.html**, servida cuando se accede a la página de inicio de sesión. Un ejemplo de su resultado final:

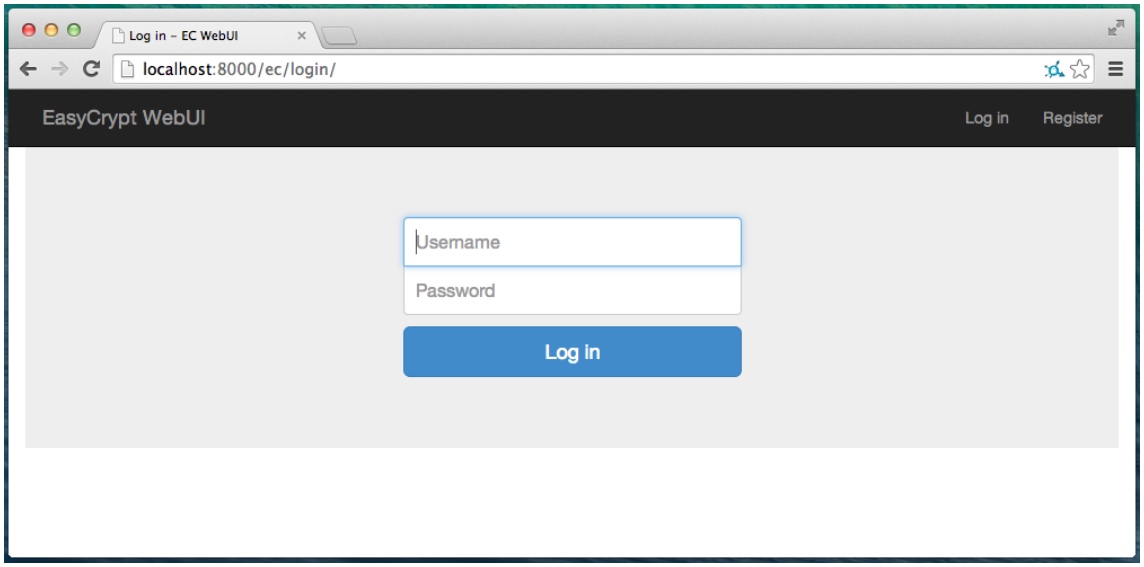


Figura 3.4: Pantalla de inicio de sesión

- **register.html**, servida cuando se accede a la página de registro de usuarios. Su resultado:

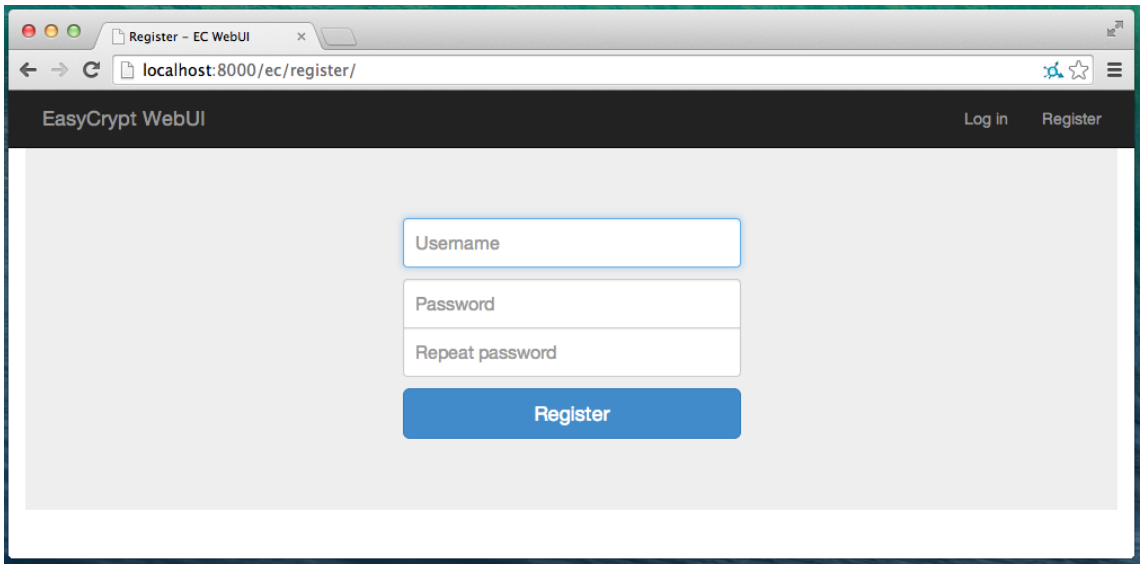


Figura 3.5: Pantalla de registro

- **index.html**, la plantilla que sirve la pantalla principal. Cuando no hay ninguna sesión iniciada, se muestra un mensaje de bienvenida:

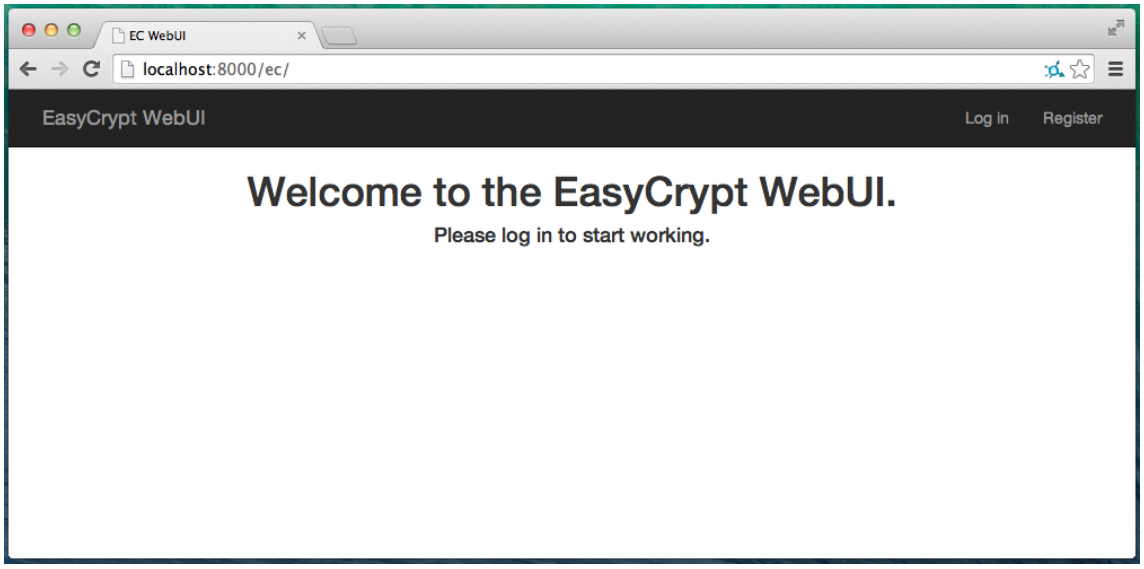


Figura 3.6: Pantalla principal (sin haber iniciado sesión)

  Una vez el usuario ha iniciado sesión, se mostrará la página principal de la interfaz web, que construiremos en la sección 3.3.

3.2.3. Controlador

El controlador es el subsistema que coordina los otras dos (modelo y vista) e implementa la lógica del sitio web como tal. En Django esto se lleva a cabo fundamentalmente en dos ficheros:

- **views.py**. En este fichero se definen las **vistas**, que es, curiosamente, el término que usa Django para denominar a las funciones que reciben una petición y devuelven una respuesta HTTP conteniendo, normalmente, una plantilla ya procesada. Este es el mecanismo básico que controla el comportamiento del sitio web: al acceder a una URI concreta se ejecuta una vista que decide, en función de los datos contenidos en la petición (sesión, usuario autenticado, método HTTP, etc.) qué plantilla servir y cómo rellenarla.

  Para implementar la funcionalidad del sitio web se han tenido que implementar varias vistas, tratando dentro de lo posible que cada vista esté asociada a un recurso, y haga una cosa u otra en función del método HTTP (arquitectura REST). Debido a que la mayor parte de la funcionalidad del sistema se encuentra en la interfaz del usuario (cliente pesado), la interacción con el servidor se reduce prácticamente al intercambio de datos: proyectos y ficheros.

- **urls.py**. En este fichero se configura qué vista ha de invocarse cuando el usuario accede a una determinada URI. El formato es bastante sencillo: las URI se especifican usando expresiones regulares, y cuando la solicitud coincide con una de ellas, se invoca la vista correspondiente pasándole como argumentos la solicitud en cuestión y, opcionalmente, argumentos especificados en la expresión regular.

  Pongamos un ejemplo real de nuestra implementación. Esta línea en urls.py se encarga de gestionar las solicitudes relacionadas con un fichero en concreto (obtener su contenido, actualizarlo, eliminarlo, etc.):

```
url(r'^files/(?P<file_id>\d+)', views.file_mgr)
```

Listado de código 3.2: urls.py

  El primer campo es la expresión regular. Los paréntesis a continuación de la barra «/» indican un campo cuyo valor es de uno o más dígitos y que se le debe pasar a la vista como argumento de nombre «file_id». En este caso, un acceso a "/files/13/" devolvería el resultado de ejecutar la vista de la siguiente forma:

```
views.file_mrg(request, file_id=13)
```

  (Que devuelve el contenido del fichero cuyo atributo "id" es 13).

  Dentro de las URI que se han diseñado para interactuar con nuestro sistema, algunas están pensadas para que el usuario pueda acceder directamente o forman parte de la navegación normal ("/login", "/logout", etc) y cuya respuesta es siempre HTML, mientras que otras se usan como interfaz directa a los datos ("/files", "/projects", etc) y devuelven información en formato JSON.

3.3. Implementación: cliente

El cliente es el segundo componente fundamental del sistema, y seguramente el más importante. El cliente interactúa con todos los agentes involucrados en el sistema: con el usuario a través del editor y el navegador de ficheros, con el servidor web a través de HTTP para mantener los datos sincronizados y con el servidor EasyCrypt para evaluar código y mostrarle de nuevo los resultados al usuario.

Como se ha mencionado anteriormente, el cliente podría clasificarse como «pesado», ya que contiene más código y tiene más carga de trabajo que el servidor a la hora de realizar la tarea fundamental de toda la interfaz web: la edición y evaluación de código. El cliente («index.js») se carga desde el servidor web al entrar en la página principal con la sesión iniciada, y posteriormente sólo se establece comunicación con el servidor web para intercambiar datos (actualización del contenido de un fichero, por ejemplo).

Para evitar tener que estar escaneando el DOM constantemente para conocer la situación actual del sistema, se ha optado por separar la lógica de la presentación. Para gestionar la lógica del cliente se ha definido un objeto JavaScript llamado «Workspace» que contiene toda la funcionalidad del editor y su estado interno. De este modo, cualquier modificación en la interfaz altera el estado del Workspace y éste, a su vez, se encarga de reflejar los cambios en el DOM. Éste es el esquema de atributos (estado) que almacena el Workspace:

```
var Workspace = function() {
  this.ui       = null;
  this.projects = [];
  this.tabs     = [];
  this.active   = null;
  this.editor   = null;

  this.ui = {};
  this.ui.treeview = $('#projects');
  this.ui.contents = $('#contents');
  this.ui.tabctl   = $('#tabs');
  this.ui.editor   = $('#editor');
}

ws = new Workspace();
```

Listado de código 3.3: Workspace (index.js)

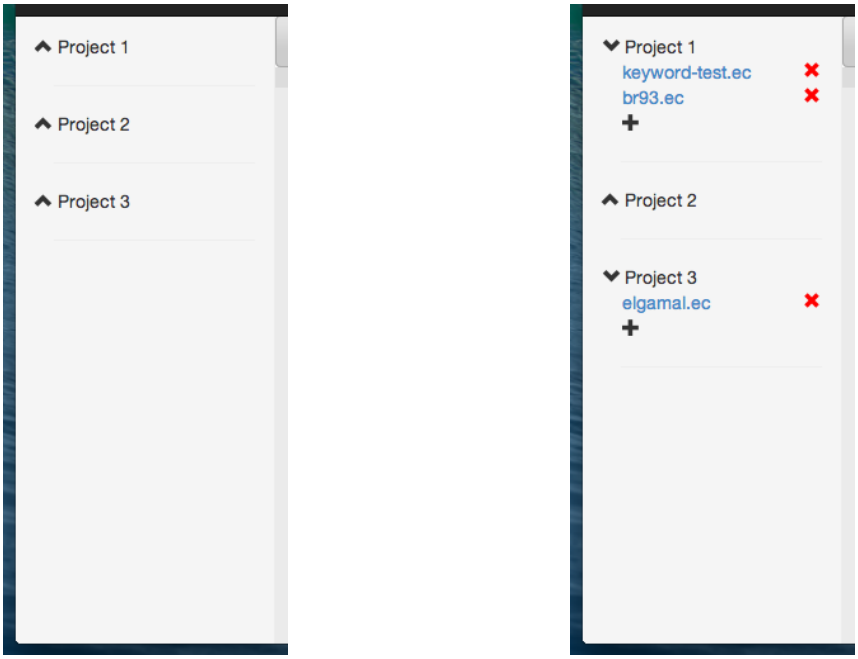
Cabe destacar la presencia del atributo «Workspace.ui», que contiene referencias a los nodos DOM con los que se modifica la parte visible del editor.

A la hora de distribuir los elementos de la página principal (de acuerdo al diseño de la figura 3.2) se ha usado una característica concreta de Twitter Bootstrap: su sistema de rejillas[1]. Bootstrap define dos elementos para trabajar con rejillas: **filas** y **columnas**. Se empieza definiendo una serie de filas que se apilarán una sobre la otra. Dentro de cada fila debe haber una o varias columnas, y a cada una de ellas se le asigna un porcentaje de la anchura de la fila contenedora. Por último, las columnas pueden contener una o más filas para seguir asignando espacio cada vez con más nivel de detalle.

Con el sistema de rejillas de Bootstrap tenemos ya la colocación básica de los elementos de acuerdo al diseño de la figura 3.2. A continuación pasamos a implementar los otros tres integrantes fundamentales de la interfaz web: el navegador de ficheros, la edición de código y la presentación de resultados.

3.3.1. Navegador de ficheros

El navegador de ficheros se ha implementado usando, de nuevo, el sistema de rejillas de Bootstrap. Cada entrada (proyecto, fichero) es una fila y se usan columnas para colocar el texto y los botones (desplegar proyecto, crear/eliminar fichero). Al hacer click en el botón de desplegar proyecto se muestran los ficheros que contiene, como se ve en la figura 3.7.



(a) Proyectos plegados (b) Proyectos 1 y 3 desplegados

Figura 3.7: Navegador de ficheros

[1] `http://getbootstrap.com/css/#grid`

La lista de proyectos y su contenido se obtiene accediendo mediante GET a la URI «/projects». Los resultados (que viajan en formato JSON) se almacenan en el atributo «projects» del Workspace, y posteriormente se muestran en la interfaz. Este proceso se repite siempre que el usuario crea o elimina un fichero o un proyecto, para garantizar la sincronización de los datos.

3.3.2. Editor de código

Cuando el usuario hace click en el nombre de un fichero se realizan varias tareas:

- Se busca el objeto «File» correspondiente al fichero
- Se crea una nueva pestaña usando el widget[1] de la biblioteca jQuery UI
- A la nueva pestaña se le asigna el fichero, una nueva sesión de editor (para guardar la posición del cursor, las modificaciones no guardadas, etc.), un título para mostrar y un enlace a su elemento correspondiente en el DOM por comodidad
- Se **activa** la pestaña, lo que significa que el editor pasa a mostrar el contenido del fichero

El comportamiento en casos similares es el que cabría esperar. Por ejemplo, en caso de que el fichero ya haya sido abierto, no se crea una nueva pestaña pero sí se activa para mostrar su contenido.

El editor de código como tal es una instancia de Ace incrustada bajo el widget de pestañas dentro del hueco asignado por la columna de Bootstrap correspondiente. Tras su inicialización, lo único para lo que se accede a él es para modificar la sesión activa cuando se cambia de pestaña. El conjunto de pestañas y editor de código se muestra en la figura 3.8.

[1] `http://jqueryui.com/tabs/`

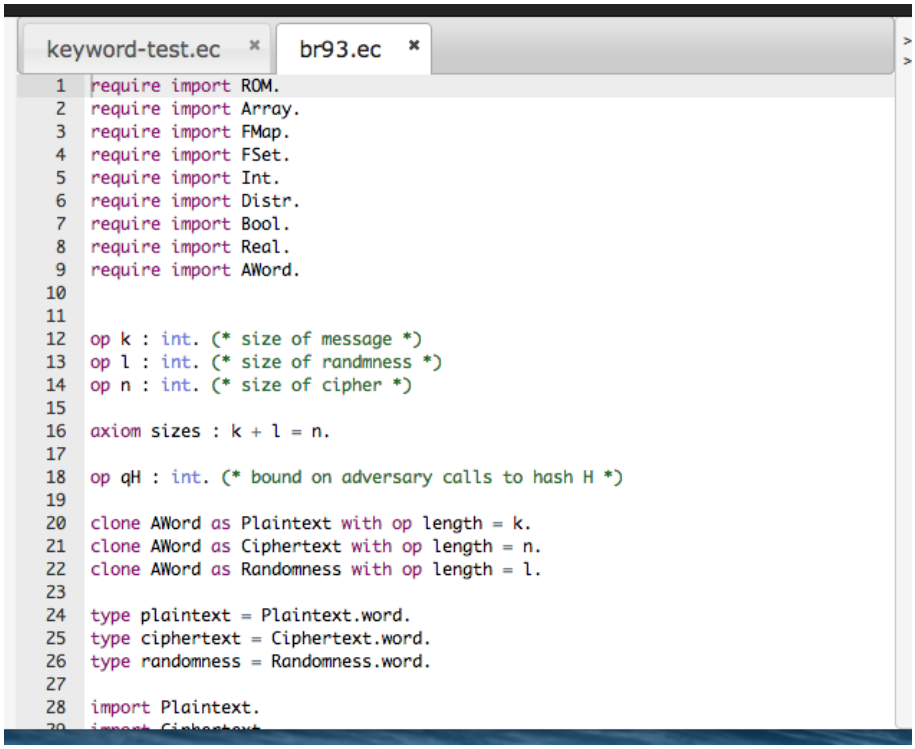


Figura 3.8: Pestañas y editor de código

El coloreado sintáctico se ha tenido que definir a mano, ya que evidentemente Ace no incluye por defecto soporte para el coloreado de EasyCrypt.

3.3.3. Presentación de resultados

El mecanismo de evaluación se basa en «pasos»: el usuario pulsa una combinación de teclas (actualmente Ctrl-Mayus-n, pero se puede modificar) para que el editor busque el siguiente punto «.», que es el caracter que EasyCrypt usa para separar sentencias. La sentencia se envía entonces mediante tecnología WebSockets al servidor EasyCrypt (sección 3.4). Una función asíncrona se encarga de mostrar a la derecha del editor (figura 3.9) todos los resultados que vayan llegando (lo cual permite, por ejemplo, seguir enviando sentencias aunque una de ellas no haya terminado de evaluarse).

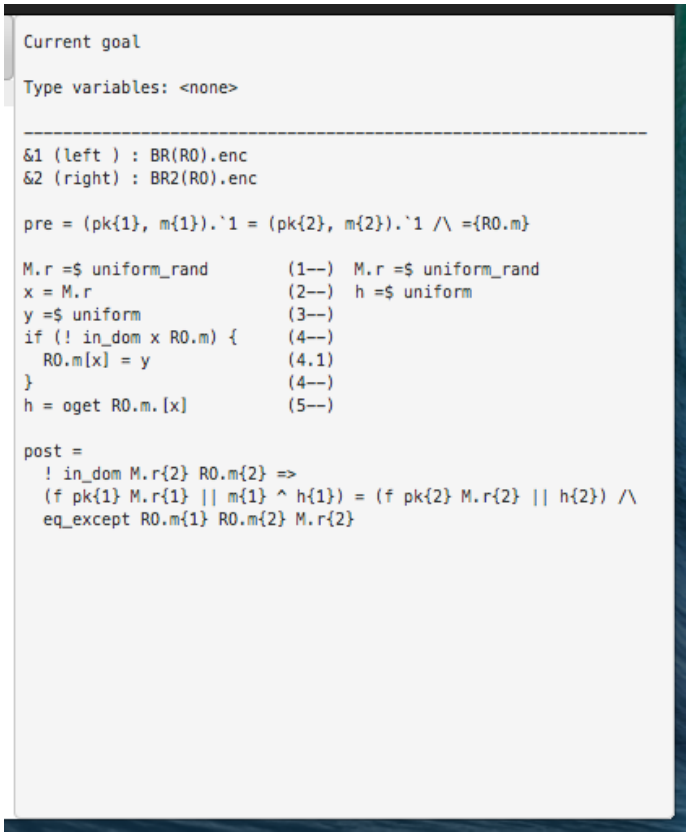


Figura 3.9: Presentación de resultados

3.4. Implementación: servidor EasyCrypt

El tercer y último componente del sistema es el servidor EasyCrypt, encargado de recibir sentencias mediante WebSockets, evaluarlas y reenviar los resultados.

Para esta tarea se ha utilizado un sencillo programa intermediario escrito en Python. El programa simplemente abre un puerto y queda a la espera de crear conexiones WebSockets en el mismo (todo esto lo implementa la biblioteca Tornado).

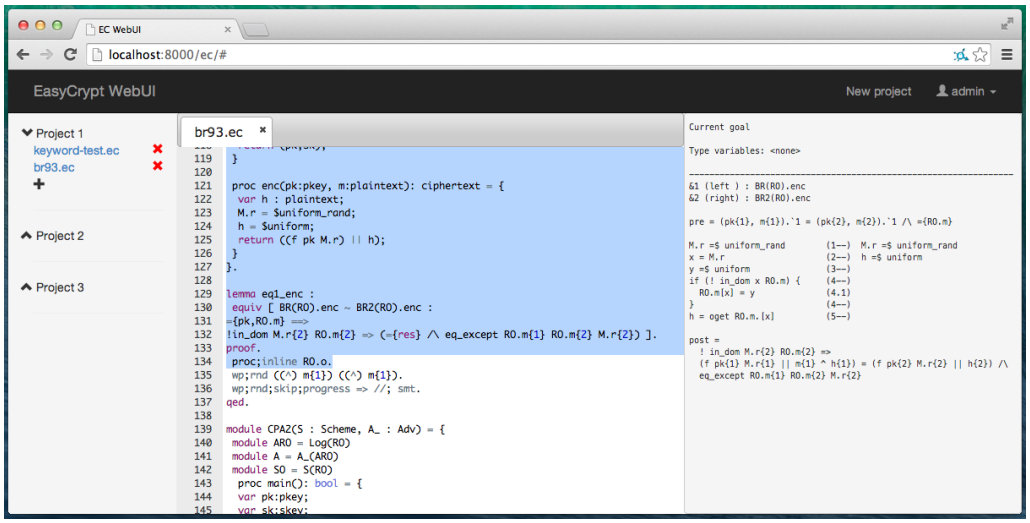
Al abrirse una conexión se ejecuta una nueva instancia de EasyCrypt y se mantiene el acceso a sus entrada y salida estándares a través de objetos Stream, de nuevo proporcionados por Tornado. El programa asigna una función asíncrona «callback» que responde al evento lanzado al haber datos disponibles ya sea en la salida estándar de EasyCrypt o en el WebSocket, y los redirige al otro extremo (WebSocket o entrada de EasyCrypt, respectivamente).

Capítulo 4

CONTRIBUCIONES

En la primera parte del documento se ha implementado la definición del coste en el lenguaje de expresiones de EasyCrypt modificando su analizador del lenguaje. Como se indica en la sección de objetivos (1.4), el siguiente paso natural es implementar tácticas y juicios que permitan usar estas definiciones para extraer conclusiones reales, ampliando considerablemente el abanico de teoremas que se puedan extraer del uso de EasyCrypt.

En cuanto a la interfaz web, se ha conseguido implementar exitosamente un sistema multiusuario capaz de gestionar, editar y ejecutar proyectos de EasyCrypt, alcanzando una funcionalidad similar a la del sistema actual (Emacs + ProofGeneral) pero evitando los costes de instalación y aprendizaje del entorno. A corto plazo se puede contar con que los profesionales dedicados a la criptografía tengan menos reparos en usar EasyCrypt como herramienta de verificación. A largo plazo, esperamos haber contribuido a la adopción de estos métodos formales que tanto podrían aportar al mundo de la seguridad.



Capítulo 5

CONCLUSIONES

A lo largo de este documento hemos abordado la mejora de dos aspectos relativamente independientes de EasyCrypt: una principalmente técnica (coste) y otra de usabilidad (interfaz web).

EasyCrypt es un marco orientado a un usuario muy concreto: profesionales de la criptografía. Si bien es cierto que en el propio Instituto IMDEA Software se usa y desarrolla activamente, el objetivo último sería crear comunidad. La verificación de algoritmos criptográficos es un área de investigación muy reciente, por lo que es necesario dedicarle recursos para que eventualmente sus contribuciones alcancen al público en general.

Si bien es cierto que el desarrollo de funcionalidad técnica es importante y contribuye al propósito con el que se concibió EasyCrypt, creo que en este caso cabe destacar la segunda parte del trabajo como la que realmente ha supuesto una innovación en el ámbito en que se mueve este tipo de software. El desarrollo de esta interfaz web supone una gran diferencia en el tipo de características que se han implementado hasta ahora en EasyCrypt, y la primera orientada exclusivamente a facilitar su uso.

Por ello, si de entre todos los conocimientos técnicos que he adquirido durante la realización de este trabajo, que no son pocos, tuviese que extraer una conclusión más general, sería que incluso los sistemas más complejos y de ámbito puramente académico necesitan usuarios, y a todo usuario le gusta que se lo pongan un poco más fácil. El tiempo dirá si ha valido la pena el esfuerzo.

Capítulo 6

ANEXOS

Todo el código desarrollado en el presente trabajo, una vez pasados los protocolos de integración, se incorporará a la versión estable de EasyCrypt y podrá obtenerse desde el sitio web (`https://www.easycrypt.info`).

Bibliografía

[1] S. Goldwasser and S. Micali, "Probabilistic encryption," *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 270–299, 1984. 2

[2] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," *IACR Cryptology ePrint Archive*, vol. 2004, p. 332, 2004. 3, 4

[3] G. Barthe, B. Grégoire, and S. Z. Béguelin, "Formal certification of code-based cryptographic proofs," in *POPL* (Z. Shao and B. C. Pierce, eds.), pp. 90–101, ACM, 2009. 3, 4, 6

[4] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *CRYPTO* (P. Rogaway, ed.), vol. 6841 of *Lecture Notes in Computer Science*, pp. 71–90, Springer, 2011. 4

[5] A. Church, "A formulation of a simple theory of types," *Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940. http://www.jstor.org/stable/2266866Electronic Edition. 5

[6] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, pp. 656–715, Oct. 1949. 12

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Thu Jun 26 23:51:39 CEST 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)